NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

TRUSTWORTHY SYSTEM DEVELOPMENT THROUGH HIGH-LEVEL SYNTHESIS

by

Isaac Patterson

September 2014

Thesis Advisor:	Theodore Huffmire
Second Reader:	Mark Gondree

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave Blank)*	**2. REPORT DATE** 09-26-2014	**3. REPORT TYPE AND DATES COVERED** Master's Thesis 01-01-2013 to 09-26-2014

4. TITLE AND SUBTITLE TRUSTWORTHY SYSTEM DEVELOPMENT THROUGH HIGH-LEVEL SYNTHESIS	**5. FUNDING NUMBERS**
6. AUTHOR(S) Isaac Patterson	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

Major processor manufacturers have embraced the high-level synthesis (HLS) design philosophy. HLS offers the potential to explore the design space of electronic circuits and systems more efficiently than traditional methods. In this thesis, we investigate the application of HLS to hardware-oriented security and trust by developing a model of a simple 16-bit Central Processing Unit in the SystemC modeling language. We enhanced our processor with a simple security mechanism that enforces a memory integrity policy. The integrity policy allows a region of the program labeled as trustworthy to modify any address in data memory, but another region of the program labeled as untrustworthy is restricted to only being able to modify a specific region of data memory. Our timing results show that adding the integrity policy enforcement mechanism has a negligible effect on overall system performance.

14. SUBJECT TERMS High-Level Syntheis, SystemC, Trustworthy System Development, Hardware-Oriented Security and Trust, Malicious Hardware, Electronic Design Automation, Electronic System-Level Design, Military Electronics, Supply Chain Security	**15. NUMBER OF PAGES** 69
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500 **Standard Form 298 (Rev. 2–89)**
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

TRUSTWORTHY SYSTEM DEVELOPMENT THROUGH HIGH-LEVEL SYNTHESIS

Isaac Patterson
Lieutenant, United States Navy
B.S., Brigham Young University, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2014

Author:	Isaac Patterson
Approved by:	Theodore Huffmire Thesis Advisor
	Mark Gondree Second Reader
	Peter Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Major processor manufacturers have embraced the high-level synthesis (HLS) design philosophy. HLS offers the potential to explore the design space of electronic circuits and systems more efficiently than traditional methods. In this thesis, we investigate the application of HLS to hardware-oriented security and trust by developing a model of a simple 16-bit Central Processing Unit in the SystemC modeling language. We enhanced our processor with a simple security mechanism that enforces a memory integrity policy. The integrity policy allows a region of the program labeled as trustworthy to modify any address in data memory, but another region of the program labeled as untrustworthy is restricted to only being able to modify a specific region of data memory. Our timing results show that adding the integrity policy enforcement mechanism has a negligible effect on overall system performance.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

HLS	high-level synthesis
NRE	nonrecurring engineering costs
NPS	Naval Postgraduate School
MSB	most significant bit
LSB	least significant bit
RAM	random access memory
ROM	read-only memory
DOD	Department of Defense
FBI	Federal Bureau of Investigation
HOST	hardware-oriented security and trust

THIS PAGE INTENTIONALLY LEFT BLANK

Executive Summary

Major processor manufacturers have embraced the high-level synthesis (HLS) design philosophy. For example, Xilinx has incorporated HLS into its Vivado suite of Electronic Design Automation (EDA) tools. HLS offers the potential to explore the design space of electronic circuits and systems more efficiently than traditional methods. The HLS design process begins with a functional model that is iteratively refined to progressively finer levels of detail, eventually resulting in a cycle-accurate model of the system. In this thesis we investigate the application of HLS to hardware-oriented security and trust (HOST) by developing a model of a simple 16-bit CPU in the SystemC modeling language. Using SystemC, designers can express both hardware and software constructs in C++; therefore, the hardware and software of an embedded system can be simulated in the same environment, rather than using separate hardware and software simulators. Only a C++ compiler and the SystemC library are needed to design and simulate a circuit.

Our processor is based on the design from the "Nand to Tetris" course that teaches computer science concepts across all levels of system abstraction by constructing a general-purpose computer system from the ground up, starting with digital logic gates and then progressing to a 16-bit CPU architecture, assembler, computer, and high-level language programming. To demonstrate the applicability of the HLS design approach to hardware-oriented security and trust, we enhanced our processor with a simple security mechanism that enforces a memory integrity policy. The integrity policy allows a region of the program labeled as trustworthy to modify any address in memory, but another region labeled as untrustworthy is restricted to only being able to modify a specific region of memory. Our timing results show that adding the integrity policy enforcement mechanism has a negligible effect on overall system performance. HLS has the potential to help designers of security enhancements as well as designers of the systems themselves, and a SystemC approach has the potential to make hardware design more accessible to computer scientists. Future work will involve exploring a wider variety of programs, policies, policy enforcement mechanisms, and processors, as well as increasing the memory size, as cycle-accurate modeling of a large number of memory cells requires a very large RAM overhead, which is a known challenge with SystemC modeling.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction and Motivation

Trustworthy system development is a major concern for the Department of Defense, which operates a large variety of complex systems that must be resilient to a wide array of developmental and operational attacks. Developing trustworthy systems is expensive due to the high non-recurring engineering (NRE) costs of developing hardware and software, and the small customer base over which to amortize that high NRE cost. In addition, system designers are increasingly concerned with the security of the entire supply chain, including hardware [1], [2] and design tools [3]. Compromised hardware has the potential to undermine policy enforcement mechanisms implemented in software. Addressing the security problem of the supply chain of electronics is very challenging due to the large number of vendors of electronic components and intellectual property (IP). Building hardware in a trusted foundry is one approach to addressing these issues, but building custom hardware in this way is costly. In addition to the fabrication costs, which increase according to Rock's law in super linear fashion, the engineering costs are also very high [4], [5]. Even using reconfigurable hardware, such as field-programmable gate arrays (FPGAs), does not necessarily reduce the design cost although it may reduce the fabrication cost.

The goal of this thesis is to reduce the cost and time for developing trustworthy hardware by leveraging high-level synthesis (HLS) to efficiently explore the design space in order to determine which design point optimally balances the tradeoffs of concern for the customer. HLS allows for the development of functional models at a high level of abstraction that can be quickly implemented in software [6]. Further refinement of a functional model results in a transactional model, and further refinement of a transactional model results in a timing model. Finally, the cycle-accurate model is the lowest level of abstraction and the finest level of granularity.

Development and simulation of a complete cycle-accurate model is too expensive for all points in the design space. Therefore, the HLS methodology relies on quickly building coarse-grained models (e.g., the functional models) to quickly determine important metrics such as power and performance at a coarse level of granularity. By allowing the designer to evaluate tradeoffs efficiently at a coarse level of granularity, HLS enables the design process to be more efficient than traditional methods. The designer can make important decisions at this stage before embarking on the tedious efforts and expensive costs of refining the coarse-grained design down to a cycle-accurate, fine-grained model. The beauty of this approach is that once the optimal point within the design space is determined, the high-level model can immediately be utilized, and the process of refinement can begin.

A major language for system modeling is SystemC, which is based on the C++ programming language. SystemC is very simple and consists of a C++ library that can be readily downloaded for free. SystemC allows a designer to express a functional model in a modified C++ language. In addition to expressing software, SystemC provides the advantage of being able to design hardware in this language. This is an improvement over traditional techniques in which software is designed in a traditional programming language, and hardware is designed in a traditional hardware description language, or HDL. The problem with the traditional approach is that the hardware is simulated in a hardware simulator, while the software is simulated in a software simulation environment. Having separate simulation environments for hardware and software is inefficient and inhibits the ability to co-design the hardware and software.

We are not the first to apply HLS to hardware trust. Bathen and Dutt developed PoliMakE, which uses HLS to explore policies for multi-core processors [7]. PoilMakE is built on top of their SystemC simulation engine. SystemC is also used in PHiLOSoftware, which helps engineers design trustworthy systems based on multi-core processors [8].

Concerns about information security for modern computers have existed nearly since their inception. With the widespread use of modern computers, the need to provide information security became more evident [9]. Saltzer and Schroeder focus on safeguarding information for systems with multiple users on the same system [9]. As malicious software emerged, including computer viruses, worms, and Trojans, patches and firewalls were developed as countermeasures. While malicious software poses a tremendous challenge, recognition of the problem of malicious hardware has emerged, as the integrated circuit supply chain is world-wide. The potential for hardware breaches has increased as global consumption relies more heavily on outsourced equipment [2]. This thesis will assess and demonstrate how high-level synthesis (HLS) can facilitate the design of policy enforcement circuitry.

CHAPTER 2: Related Work

The fields of computer security and computer hacking have evolved over time. Just as programmers work to protect system security, skilled and motivated hackers will attempt to exploit weaknesses in the protection mechanisms.

Multiple publications touch on the matter of safeguarding computer systems. One of the most seminal of these works is J. H. Saltzer and Michael D. Schroeder's "The Protection of Information in Computer Systems," written in 1975 [9]. The authors of this work discuss how the invention of the Von Neumann general-purpose architecture drastically reduced the production cost of modern computers, which allowed wide spread use of the machines. Saltzer and Schroeder wrote their work with the "key concern" of safeguarding information against multiple users on the same system. As computer users and designers get savvier in their attempts to prevent software security breaches, malicious hackers must find new areas to attack, where advanced security has not yet been implemented. A more recent publication entitled "Trustworthy Hardware: Identifying and Classifying Hardware Trojans" addresses the possibility of security threats introduced at the hardware level of modern computing. The potential for hardware breaches has increased as global consumption relies more heavily on outsourced equipment [2].

Karri et al. survey the emerging discipline of hardware oriented security and trust [2]. In a world in which hackers develop sophisticated exploits, they devise novel ways to bypass security mechanisms. Hardware vulnerabilities represent a means for such an exploit to occur. Karri et al. present a taxonomy of malicious circuitry for classifying malicious inclusions and countermeasures.

The Department of Defense (DOD), like the rest of the information technology world, finds itself reliant on global outsourcing for the manufacturing of digital infrastructure. Therefore, the DOD is interested in mitigating supply chain threats by using enhanced government services, encouraging improved commercial practices, and requiring supply chain risk management [10]. While the NSA has established dozens of trusted foundries, manufacturing all military electronics in trusted foundries may not be feasible. Design and manufacture of all hardware and software intellectual property in house is expensive, time consuming, and might not yield a defect-free result [2]. The vulnerability exists for untrusted foundries to maliciously modify a circuit without user knowledge.

A three-year investigation conducted by the Federal Bureau of Investigation (FBI), from 2004-2006, discovered "counterfeit Cisco routers in U.S. defense, finance, and university networks" [11]. Even more alarming than the actual discovery of the compromised hardware was the fact that many of the routers came directly from "untrustworthy sources in foreign countries." While the investigation did not detect malicious hardware injections and concluded that the infiltrator's motivations appeared merely fiscal, the investigation "vividly illustrate[d] the vulnerability" users face when purchasing unverified hardware [2, p. 39].

Karri et al. suggest the creation of a "hardware Trojan taxonomy" to address the possible introduction of malicious circuitry made in "untrusted factories" [2]. They state, "To be trustworthy, hardware must exhibit only the functionality for which it was designed, nothing more and nothing less; conceal any information about the computation performed through any side channels such as power and delay; and be transparent only to the designer while remaining opaque to others, who should know nothing about its design and internal states" [2]. They also provide a more detailed definition of a hardware Trojan as "a malicious and deliberately stealthy modification made to an electronic device such as an IC. It can change the chip's functionality and thereby undermine trust in the systems using that trojaned chip" [2].

The hardware Trojan taxonomy created by Karri et al. is based on five different categories: the insertion phase, abstraction level, activation mechanism, effects, and location [2]. The *insertion* phase covers all possible points at which a hacker could maliciously alter hardware and remain undetected throughout the testing cycle. The *activation mechanism* phase deals with potential points at which a hardware Trojan could be activated. For example, the hardware Trojan could be either "always on" or "triggered" by an external event. The *effects* category addresses four generalized results the Trojan could create; it could "change the functionality, downgrade performance, leak information, [or] den[y] service" [2]. With the hardware Trojan taxonomy established, the real work–identifying and preventing hardware Trojan implementation–can begin.

A follow-on article entitled “Trustworthy Hardware: Trojan Detection and Design-for-Trust Challenges” delves even further into installing safeguards against potential malicious hardware elements [1]. The reliance of globalization and outsourcing of the semiconductor industry is again cited for creating the increased vulnerability of hardware Trojans [1]. To offset this challenge, the authors suggest “either the developer must make the IC design and fabrication process trustworthy or the client must verify the IC for trustworthiness” [1].

The trust element of chip manufacturing is a primary concern for both the Department of Defense and the general consumer market as a whole. In addition to trustworthiness of design, cost, performance, and functionality are important considerations in creating a “winning design.” Kurt Keutzer states, “The overall goal of electronic embedded system design is to balance production cost with development time and cost in view of performance and functionality considerations. Manufacturing cost depends mainly on the hardware (HW) components of the product” [4].

Our work explores the potential benefits of HLS for trustworthy system development. Major chip manufacturers such as Xilinx and Intel have adopted HLS in their design practices, and we argue that HLS can also facilitate the design of policy enforcement mechanisms. To validate our hypothesis, we construct a general-purpose computer in the SystemC language and enhance it with a simple memory integrity policy enforcement mechanism. We then evaluate system performance both with and without the security enhancement. While our work falls within the scope of hardware-oriented security and trust, we do not claim to directly address the problem of malicious hardware inclusions in our work.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Design Flow

In our efforts to demonstrate the capability of utilizing HLS in the design of policy enforcement circuity, we chose to follow the coursework of Nisan and Schocken [12]. The Nisan and Schocken text is complemented with online material found at http://www.nand2tetris.org [12]. The course is often simply referred to as "nand2tetris." The premise of their work is to help both individuals with and without computer science backgrounds to comprehend the process of building a modern computing machine, as well as to implement software to run on the machine. Our project focuses on the first portion of their text and coursework—the construction of the modern computer.

The Nisan and Schocken course [12] incrementally builds upon logic gates to construct a 16-bit modern computer—named HACK. Modern computers are built with transistors that physically implement simple Boolean functions, called logic gates. One of the most fundamental logic gates, described in more detail in the section below, is the NAND gate. Logically, the NAND gate performs two functions. First, it takes two inputs and performs the AND function on them. The truth table produced from this function is shown as Table 3.1. The next logic operation is the NOT function, which yields the inversion of the original input. The truth table for the NOT function is provided in Table 3.2. Combining both of these functions together produces the results shown in Table 3.3. As explained below, NAND gates are universal building blocks for combinational circuitry.

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

Table 3.1: The truth table for the AND function of discrete math.

The HACK computer is capable of performing simple 16-bit operations. Ultimately, there are 28 functions the ALU can compute (a complete list of the 28 computations is found on page 67 of Nisan and Shocken's book). This chapter follows the development of each

IN	OUT
0	1
1	0

Table 3.2: The NOT function's truth table.

A	B	(AB)'
0	0	1
0	1	1
1	0	1
1	1	0

Table 3.3: The NAND gate's truth table.

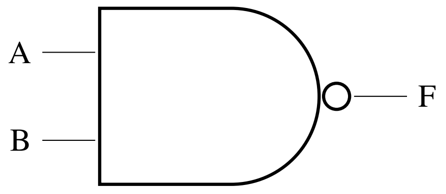
logic gate in the manner described by Nisan and Shocken, and describes how it relates to the overall development of the HACK computer. By the end of this chapter, all logic gates needed to construct the HACK machine have been created and connected to yield the HACK machine in SystemC.

3.1 "In the beginning, there was NAND..."

Because of their universality, NAND gates are fundamental building blocks of all combinational circuits. Electrical engineers can easily build NAND gates out of only a handful of transistors. All other logical gates can be constructed using NAND gates.

We created the NAND gate in SystemC by declaring two boolean inputs *A* and *B*. The NAND gate logically "ands" the two boolean inputs then negates that result, which produces the output. In the SystemC code used for this project, *F* represents the final boolean output of the NAND gate.

To properly construct and run the NAND gate in SystemC, the boolean inputs *A* and *B* are logically "anded" together, and then their result is "notted" via the function'*do_nand2.*' Figure 3.1 illustrates the composition of the NAND gate.



NAND gate

Figure 3.1: The "NAND" gate is a fundamental building block for digital logic designs.

The following is the actual SystemC code for the NAND gate:

```
SC_MODULE(nand2)          // declare nand2 sc_module
{
  sc_in<bool> A, B;       // input signal ports
  sc_out<bool> F;         // output signal ports

  void do_nand2()         // a C++ function
  {
    F.write( !(A.read() && B.read()) );
  }

  SC_CTOR(nand2)          // constructor for nand2
  {
    SC_METHOD(do_nand2);  // register do_nand2 with kernel
    sensitive << A << B;  // sensitivity list
  }
};
```

3.2 Next came ... AND

All additional logic gates required to build a digital computer can be derived from NAND gates. An AND gate in SystemC consists of two NAND gates wired together. The AND gate handles two boolean inputs *A* and *B*, uses one internal boolean signal *S1*, and produces a boolean output *F*. The SC_CTOR sets up the digital layout of the logic device by feeding *A* and *B* into the first NAND gate, *n1*. The output of *n1*, *S1*, is then fed as both the *A* and

B inputs to the second NAND gate, *n2*. This procedure yields a final output *F* for the AND gate. Figure 3.2 illustrates the composition of the AND gate:

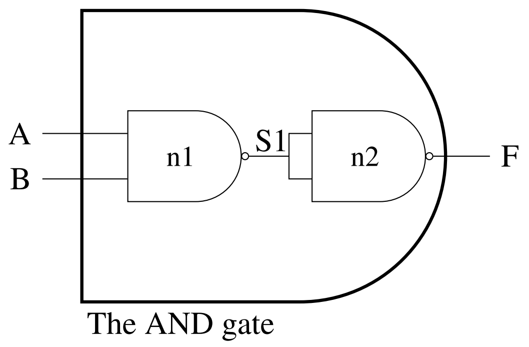


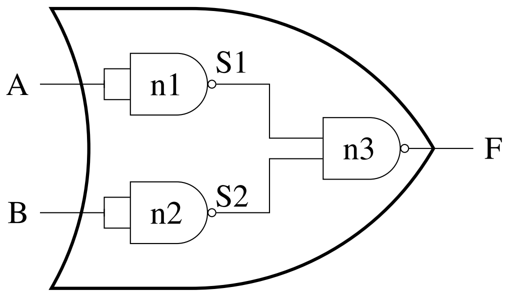
Figure 3.2: The AND gate utilizes two NAND gates to produce its output.

Here is the SystemC code for an AND gate:

```
SC_MODULE(_and2)
{
  sc_in<bool> A, B;
  sc_out<bool> F;

  nand2 n1, n2;

  sc_signal<bool> S1;

  SC_CTOR(_and2) : n1("N1"), n2("N2")
    {
      n1.A(A);
      n1.B(B);
      n1.F(S1);

      n2.A(S1);
      n2.B(S1);
      n2.F(F);
    }
};
```

3.3 Many more logic gates are now possible

We construct an *OR* gate in SystemC by utilizing three NAND gates, *n1*, *n2*, and *n3*. The logical design created utilizing the SC_CTOR accepts two boolean inputs, *A* and *B*, generates two internal signals, *S1* and *S2*, and produces a final boolean output, *F*. The first NAND gate *n1* uses *A* for both its inputs and outputs *S1*. The second NAND gate *n2* uses *B* for both its inputs and produces signal *S2*. The third NAND gate *n3* accepts *S1* and *S2* as inputs, yielding *F* as the final output. Figure 3.3 illustrates the composition of the OR gate.



The OR gate

Figure 3.3: The OR gate employs three NAND gates to produce output *F*.

Here is the SystemC code for the OR gate:

```
SC_MODULE(_or2)
{
  sc_in<bool> A, B;
  sc_out<bool> F;

  nand2 n1, n2, n3;

  sc_signal<bool> S1, S2;

  SC_CTOR(_or2) : n1("N1"), n2("N2"), n3("N3")
    {
      n1.A(A);
      n1.B(A);
      n1.F(S1);
      n2.A(B);
      n2.B(B);
      n2.F(S2);

      n3.A(S1);
      n3.B(S2);
      n3.F(F);
    }
};
```

3.4 Have you ever dealt with a NOT?

Constructing a NOT gate in SystemC merely requires one internal NAND gate, *n*. *IN* becomes both inputs to the NAND gate *n*. The output from *n* yields the final output *F*. Figure 3.4 illustrates the composition of the NOT gate.

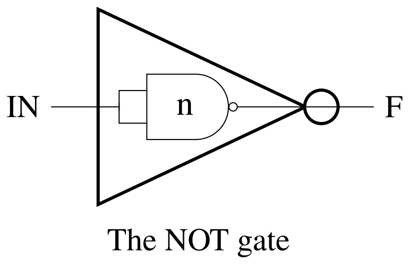


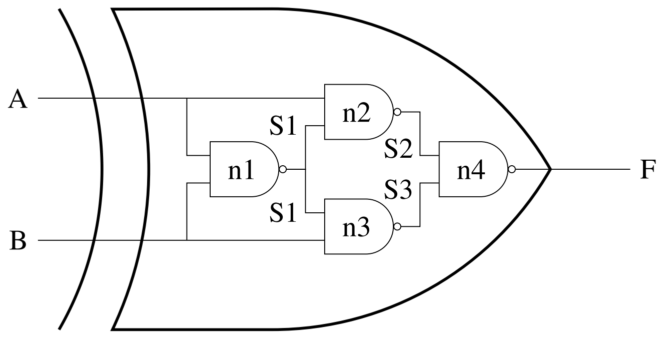
Figure 3.4: The NOT gate merely requires one NAND gate.

Here is the SystemC code for the NOT gate:

```
SC_MODULE(_not1)
{
  sc_in<bool> IN;
  sc_out<bool> OUT;

  nand2 n;

  SC_CTOR(_not1) : n("N")
  {
    n.A(IN);
    n.B(IN);
    n.F(OUT);
  }
};
```

3.5 The XOR gate

Creating an XOR gate in SystemC requires four NAND gates, *n1*, *n2*, *n3*, and *n4*. We create the logical circuit by connecting two boolean inputs (*A* and *B*) to the NAND gates as shown in Figure 3.5. In addition to the two boolean inputs, the XOR circuit uses three internal boolean signals and produces the final boolean output (*F*).



The XOR gate

Figure 3.5: The XOR gate requires four NAND gates.

3.6 The Multiplexor (a.k.a "Mux")

To build a multiplexor (aka *Mux*) in SystemC, we utilize the logic gates described above. The Mux requires two AND gates, one NOT gate, one OR gate, and three input booleans (*A*, *B*, and *SEL*). It uses three internal signals, *S1*, *S2*, and *NOTSEL*, to produce a final output *F*. Figure 3.6 illustrates the composition of a *Mux*.

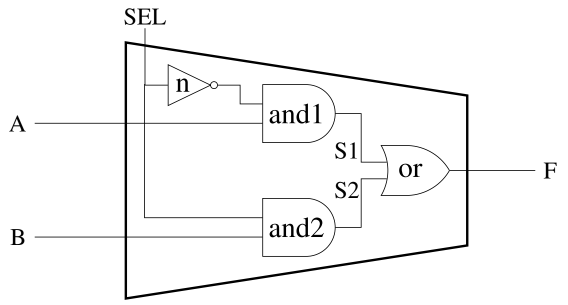


Figure 3.6: The Mux consists of two AND gates, one NOT gate, and one OR gate

3.7 D-Mux that, please

A demultiplexer, or *D-Mux*, performs the inverse function of the multiplexor. Rather than selecting between two inputs, the D-Mux "is an *output selector* which has a single input and directs it to one of **N** outputs" [13].

Constructing a *D-Mux* requires the use of two AND gates, *a1* and *a2*, and one NOT gate, *n*. The SC_CTOR wires the logic design as follows: external boolean input *IN* is wired as one of the inputs required for both *a1* and *a2*. An additional external value, *SEL* is wired directly to AND gate *a2* as its second required input. SEL is also run through the NOT gate *n* where its result, *NOTSEL*, is used as the second input for AND gate *a1*. The output of *a1* is *A*. The output of *a2* is *B*. Depending on the value of the SEL bit, the initial input value *IN* will be passed through as output *A* or output *B*. Figure 3.7 illustrates the composition of a demultiplexer.

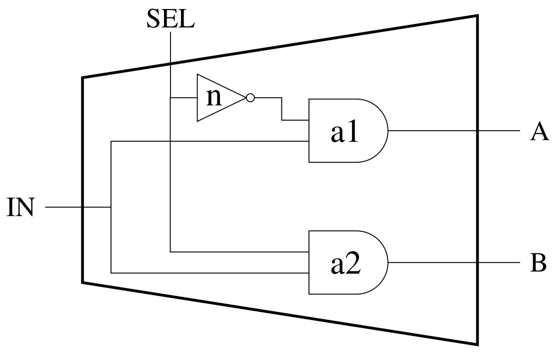


Figure 3.7: The D-Mux consist of two AND gates and one NOT gate.

3.8 Handling larger input arrays

Each of the previously described elementary logic gates in our digital design handles single-bit inputs, but for the machine we are building, 16-bit buses must be dealt with. By combining multiple single-bit gates, the 16-bit input buses can be processed. In the case of processing a 16-bit value through the NOT gates, we combine sixteen NOT gates to form a *NOT_16* gate. The composition of the *NOT_16* circuit is illustrated in Figure 3.8.

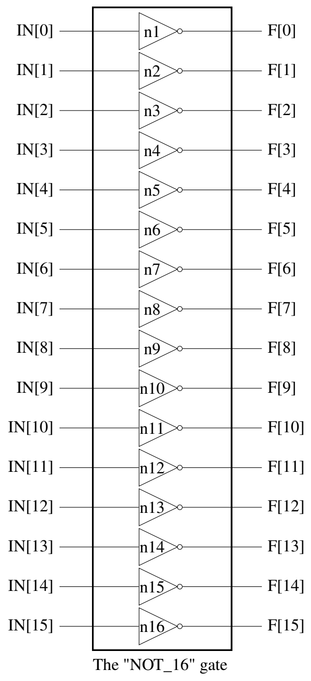


Figure 3.8: Sixteen NOT gates are placed together to handle a 16-bit input bus.

A simpler visual display is provided in Figure 3.9. In this diagram, the sixteen bits are all fed into the NOT_16 gate simultaneously, producing a 16-bit output, *F/16*.

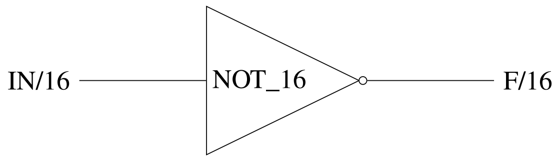


Figure 3.9: The NOT_16 gate handles a 16-bit bus input, negates the value of each simultaneously, and yields output F/16.

To complete a fully functioning simple modern computer, we continued to build additional logic chips as specified by Nisan and Schocken. The computer they designed is called the HACK machine, and we largely follow their designed machine—only creating it in SystemC instead of HDL. Just as we did with the NOT_16 gate, we combine other smaller logic gates to form logic circuits that can handle 16-bit bus inputs and produce 16-bit bus outputs. This effort quickly produced the following three logic gates: the *AND_16* gate, the *OR_16* gate, and a *Mux_16* gate. They are illustrated in Figures 3.10, 3.11 and 3.12.

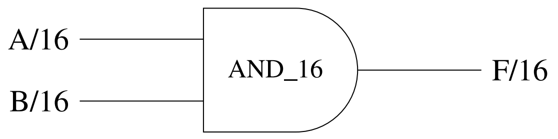


Figure 3.10: The AND_16 gate handles 16-bit bus inputs, "AND-ing" each of the input bits simultaneously

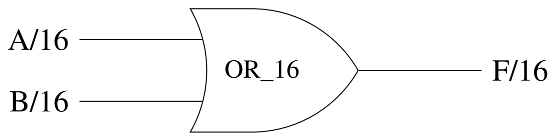


Figure 3.11: The OR_16 gate handles 16-bit bus inputs simultaneously and produces the output F/16.

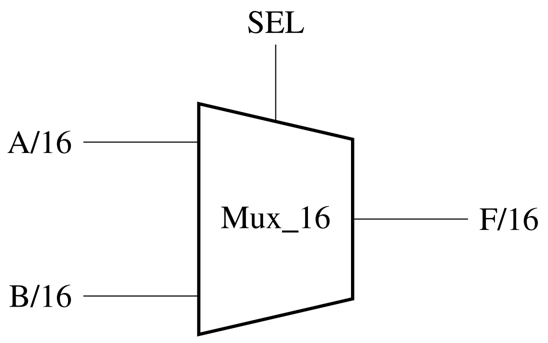


Figure 3.12: The Mux_16 gate handles two 16-bit bus inputs simultaneously to select a final output of F/16.

An additional logic circuit named the *OR_8* gate utilizes seven regular *OR* gates to select between eight boolean inputs, *A*, *B*, *C*, *D*, *E*, *F*, *G*, and *H*. The internal OR gates are *or1*, *or2*, *or3*, and *or4*. Their outputs are wired to *or5* and *or6*. Finally, the outputs of *or5* and *or6* are input into *or7*, which yields the final output *F*. The logical implication of this circuit is as follows: if any of the inputs is true, the output of the *OR_8* will be true. The only time the *OR_8* gate will produce a false or zero output is when all input booleans are also false. Figure 3.13 illustrates the internal composition of the OR_8 gate.

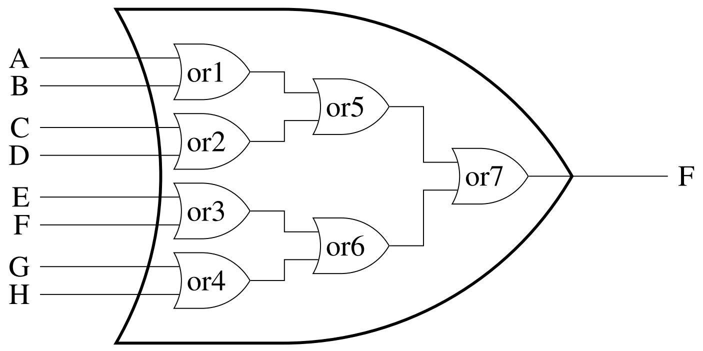


Figure 3.13: The OR_8 gate produces a "true" bit output if any of the eight input bits are true. If all eight boolean inputs are "false" or zero values, the output of the OR_8 will be zero.

3.9 Continuing the construction of larger logic gates

The next logic gate we designed in SystemC was the *Mux4way16* gate. The *Mux4way16* logic gate performs the function of selecting between four 16-bit bus inputs - *A/16*, *B/16*, *C/16*, and *D/16*. Also required in this effort are two *select* bits, *SEL0* and *SEL1*. These select bits allow a selection to be made between the four options, which yields the final output *F_16*. Internally, two additional 16-bit buses, *S/16* and *T/16*, are required. Figure 3.14 illustrates the internal composition of the Mux4way16 gate.

Progressing sequentially in our design, the next logical step was creating a *Mux8way16* gate. Just as the name suggests, the Mux8way16 gate selects a final output bus from eight 16-bit input buses. The construction of the Mux8way16 in SystemC requires eight input buses, named *A/16*, *B/16*, *C/16*, *D/16*, *E/16*, *F/16*, *G/16*, and *H/16*, and three *select* bits (*SEL0*, *SEL1*, and *SEL2*). Two *Mux4way16* gates, *m0* and *m1*, and one *Mux_16* gate, *m2*, are utilized internally to produce the desired outcome. The composition of the *Mux8way16* is illustrated in Figure 3.15.

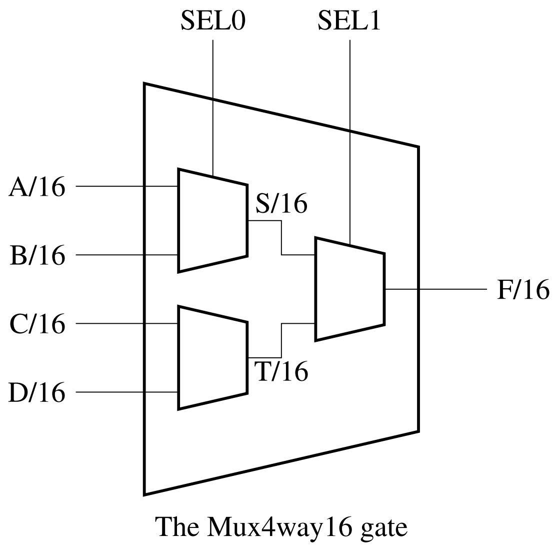


Figure 3.14: The Mux4way16 selects between four 16-bit input buses and produces one 16-bit output bus.

3.10 Once you Mux, it's easy to D-Mux

Just as we expanded the multiplexors to handle more inputs, demultiplexors can also be expanded to handle more outputs. A demultiplexer performs the inverse function of the multiplexor, taking a single input, *IN*, and directing it to one of **N** outputs. Figure 3.16 demonstrates a 4-bit *D-Mux*. Our SystemC implementation utilizes three *D-Mux* gates, *d1*, *d2*, and *d3*, and two select bits, *SEL0* and *SEL1*, to construct the *D-Mux4way*. Two internal booleans, *S* and *T*, are also used. The four final outputs are labeled *A*, *B*, *C* and *D*. The composition of the *D-Mux4way* is illustrated in Figure 3.16.

To extend the capabilities of the demultiplexer to handle eight outputs, we created the *Dmux8way* gate. The *Dmux8way* consists of one *D-Mux2way* gate, *d1*, and two *D-Mux4way* gates, *d2* and *d3*. Three input selectors, *SEL0*, *SEL1* and *SEL2*, are also required. The eight outputs are labeled *A*, *B*, *C*, *D*, *E*, *F*, *G*, and *H*. Figure 3.17 shows the logical construction of an 8-way demultiplexer.

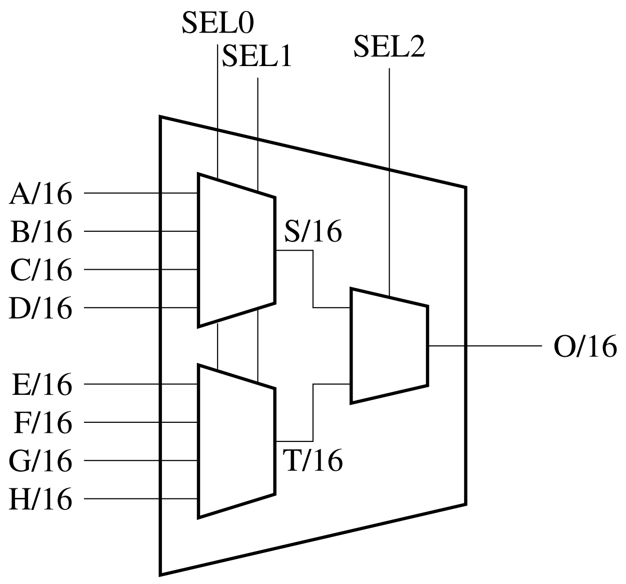


Figure 3.15: The Mux8way16 selects its final outcome choice *O/16* from among eight 16-bit buses.

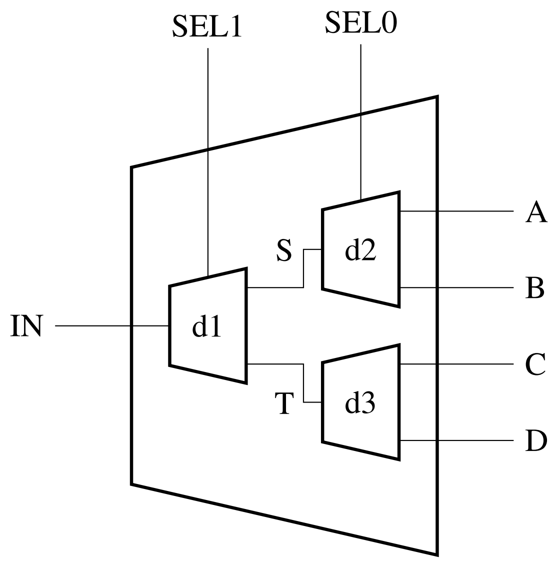


Figure 3.16: The D-Mux4way directs an incoming bit to one of four outputs.

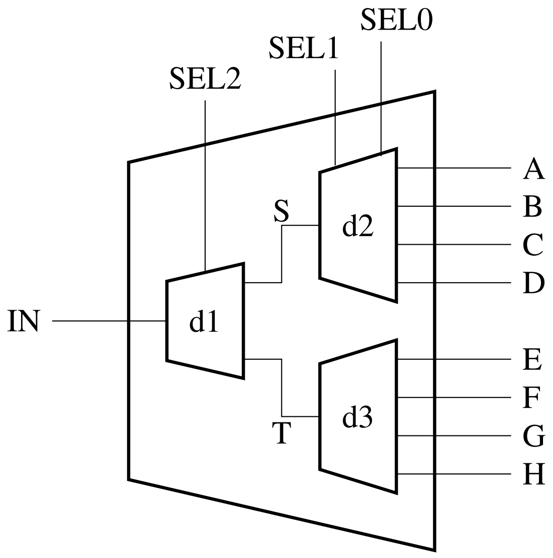


Figure 3.17: The D-Mux8way directs an incoming bit to one of eight outputs.

3.11 Time for some simple addition: Introducing the Half-Adder and Full-Adder

As we move forward with the construction of the HACK machine using SystemC, performing arithmetic operations is necessary. For the simplest arithmetic, addition, we are able to implement this capability in two steps: constructing a *HalfAdder*, followed by the construction of a *FullAdder*.

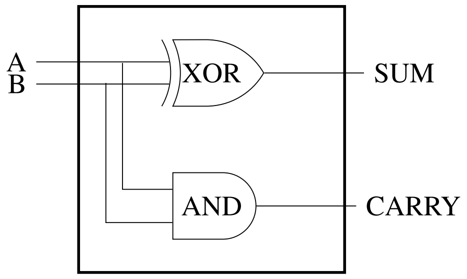


Figure 3.18: The HalfAdder

To make the *HalfAdder* in SystemC, two boolean inputs, *A* and *B*, are required, and the operation yields two boolean outputs, *SUM* and *CARRY*. Internal composition of the Half-

Adder gate requires one *XOR* gate and one *AND* gate. Their composition is shown in Figure 3.18.

Now that the *HalfAdder* has been constructed, assembling the *FullAdder* becomes possible. The *FullAdder* is built by assembling together two *HalfAdders* (*ha1* and ha2) and an *OR* gate (*or*) and then routing in three boolean inputs (*A*, *B* and *CIN*), utilizing three internal boolean signals (*S*, *T*, and *U*), and ultimately producing two boolean outputs - *SUM* and *COUT*. The diagram in Figure 3.19 illustrates the assembly of the *FullAdder*.

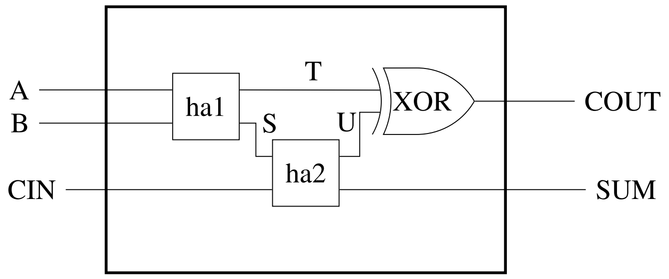


Figure 3.19: The FullAdder

3.12 Preforming addition on input buses

Now equipped with the ability to preform addition, our next goal consisted of performing addition on 16-bit values. Much like we did with previous implementations of logic chips for larger buses, the construction of the *ADD_16* gate contains sixteen FullAdders in order to process the operands.

The parts of the ADD_16 gate are arranged as follows using SystemC: two 16-bit input buses, *A/16* and *B/16*, the operands to be added together. Each of the individual FullAdder gates - *fa0*, *fa1*, *fa2*, *fa3*, *fa4*, *fa5*, *fa6*, *fa7*, *fa8*, *fa9*, *fa10*, *fa11*, *fa12*, *fa13*, *fa14*, and *fa15* - generates an internal carry-out signal (*c-out[n]*), which is passed to the next sequential FullAdder gate. The last carry-out bit is discarded. The SUM is output from each internal FullAdder then placed on a bus. Figure 3.20 illustrates the composition of the ADD_16 circuit.

Continuing our efforts to enhance the HACK machine's ability to perform arithmetic operations, implementing an *incrementor* was the next logical step. Our incrementor, named

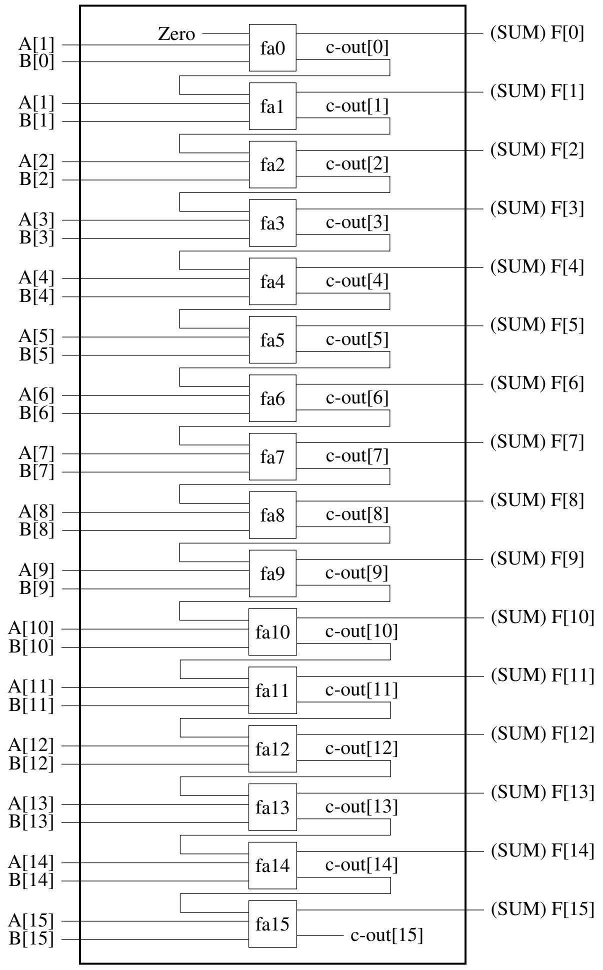


Figure 3.20: The ADD_16 gate adds two 16-bit values together.

INC_16, is designed to handle an arbitrary 16-bit boolean value and increase (or increment) it by one. Designing the *INC_16* gate in SystemC requires one *ADD_16* gate. An abstraction of the implementation of INC_16 is shown in Figure 3.21.

Another logic gate needed for the HACK machine is the *Controlled_Zero16*. The *Controlled_Zero16* circuit accepts any 16-bit input, *IN/16*, and proceeds, when instructed by the select bit *C*, to zero all bits of the bus, producing an output bus, *OUT/16*, of all zeros. If the *C* select bit is not asserted, the inputted value will be outputted as OUT/16 without any alteration to the value. To build the *Controlled_Zero16* circuit in SystemC, one *AND16* gate, one *Mux16* gate, and one *NOT/16* are utilized. Two internal boolean buses, *S/16* and

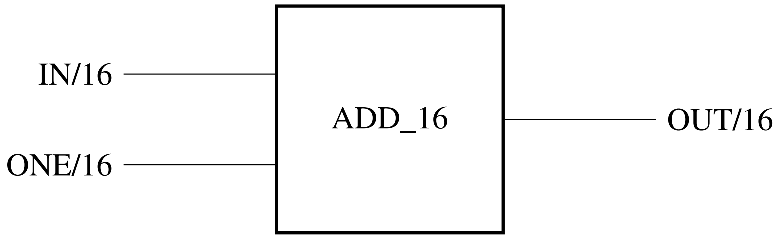


Figure 3.21: The INC_16 circuit increments an inputted value, *IN*, by one.

T/16, are also used. The logical layout of the *Controlled_Zero16* circuit is illustrated in Figure 3.22.

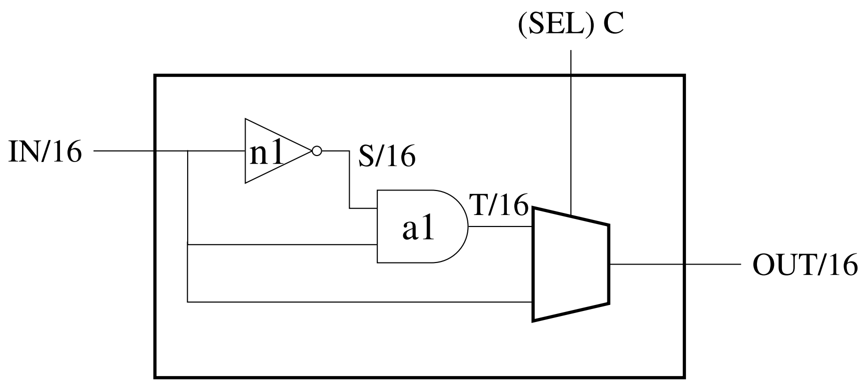


Figure 3.22: The Controlled_Zero16 gate can either zero out the entire inputted value or output the original input unaltered.

The next digital logic circuit we designed in SystemC is the *Controlled_Not* gate. Very similar to the *Controlled_Zero16* circuit, the *Controlled_Not* gate negates (or "flips") each of the 16-bit values it receives. The *Controlled_Not* circuit is constructed with one *Mux16* and one *Not16* gate. The 16-bit input bus *IN* is spliced in two directions - one running through the NOT16 gate, *n1*, before being fed into the Mux16, and the other being fed directly into the Mux16. The select bit *C* choses which value to output as *OUT/16*, either the original inputted value or its negated/complemented value. The construction of the Controlled_Not16 logical circuit is illustrated in Figure 3.23.

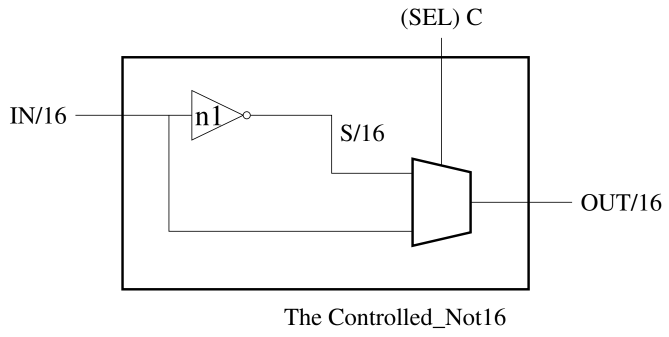


Figure 3.23: The Controlled_Not16 will either flip each bit of the input bus or output the inputted value unaltered.

3.13 Two more arithmetic circuits must be constructed prior to the ALU

As we near the ability to construct an ALU for our HACK machine, only two more arithmetic circuits are needed. The first of these two is called the *AND_or_ADD16* circuit. The AND_or_ADD16 circuit requires one *AND16* gate, one *ADD16* gate, and one *Mux16* gate. Inputs, *A/16* and *B/16* are wired to both the AND16 gate and the ADD16 gate. The outputs from the AND16 and the ADD16 are then inputted into the Mux16. The select bit *C* determines which of the two inputted values is selected for the output *OUT/16*. If the *C* bit is zero ("0"), the output of the AND16 gate is passed through as the output. Otherwise, if the *C* is a one ("1"), the output of the ADD16 is passed through as the final result, *OUT/16*. Figure 3.24 illustrates the logical arrangement of the AND_or_ADD16 circuit.

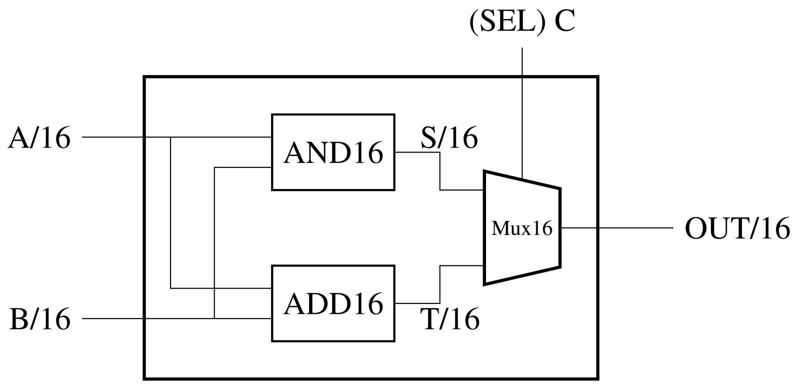


Figure 3.24: The AND_or_ADD16 circuit

The last logic circuit required before implementing the ALU is the *check_16*. The *check_16* produces a 16-bit boolean output bus, as well as two single-bit boolean output signals, *ZR* and *NG*. To construct the *check_16* circuit in SystemC, fifteen *OR* gates, sixteen *AND* gates, and one *NOT* gate are used (alternatively, two Or8way gates, one two-way OR gate and one NOT gate can also be used). The internal wiring is illustrated in Figure 3.25.

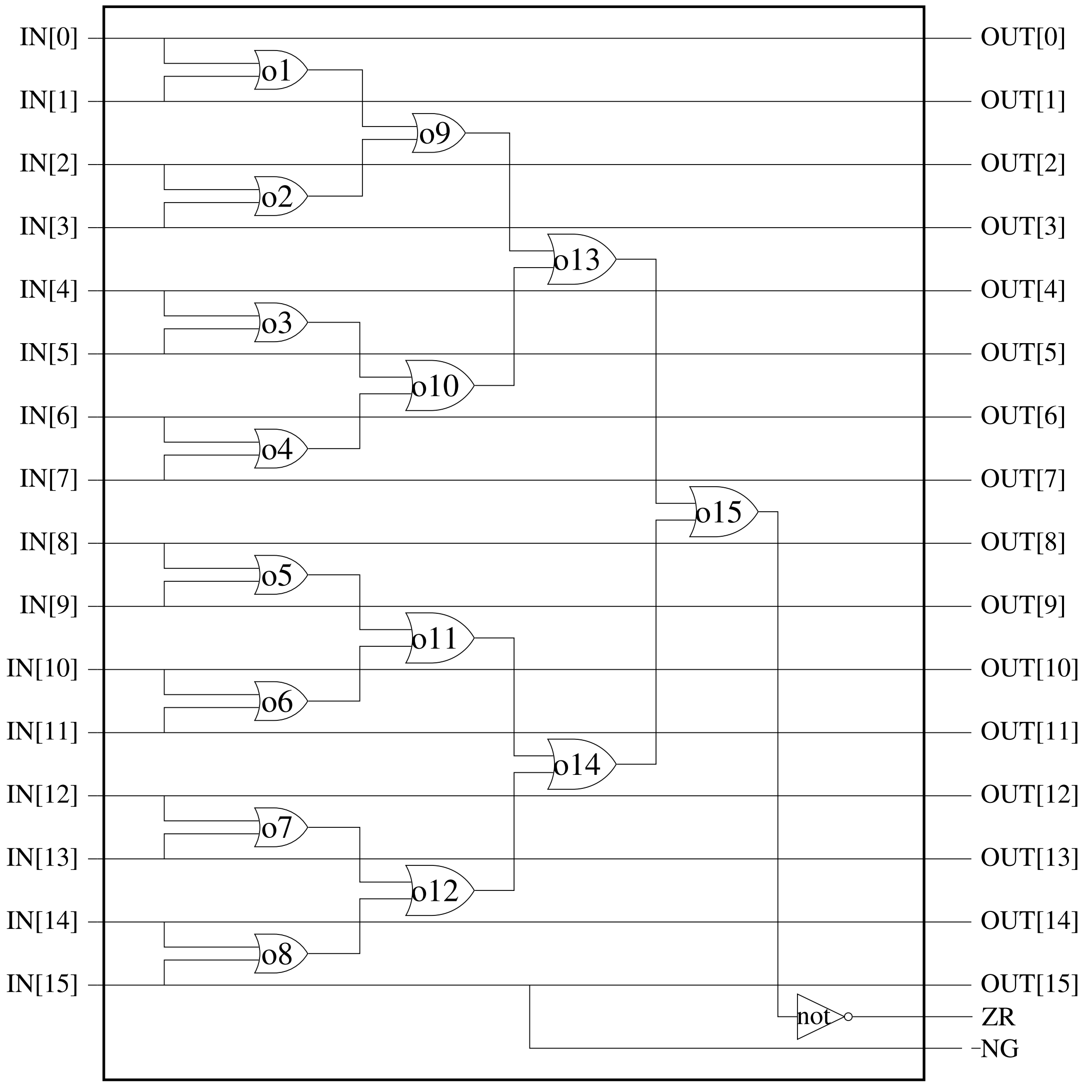


Figure 3.25: The Check_16 circuit

3.14 Time to build the ALU

We now have all digital logic tools required to build the HACK's ALU (Arithmetic Logic Unit). While "The centerpiece of the computer's architecture is the CPU... the centerpiece of the CPU is the ALU, or Arithmetic-Logic Unit" [12].

The ALU executes all the arithmetic and logical operations performed by a computer. Depending on what operations a computer designer wants his/her machine to calculate, the exact operations of the ALU may vary from one computer design to another. In our case, we are designing the HACK machine as specified by Nisan and Schocken. Nisan and Schocken state, "The Hack ALU computes a fixed set of functions *out* = *fi(x, y)* where *x* and *y* are the chip's two 16-bit inputs, *out* is the chip's 16-bit output, and *fi* is an arithmetic or logical function selected from a fixed repertoire of eighteen possible functions. We instruct the ALU which function to compute by setting six input bits, called *control bits*, to selected binary values" [12].

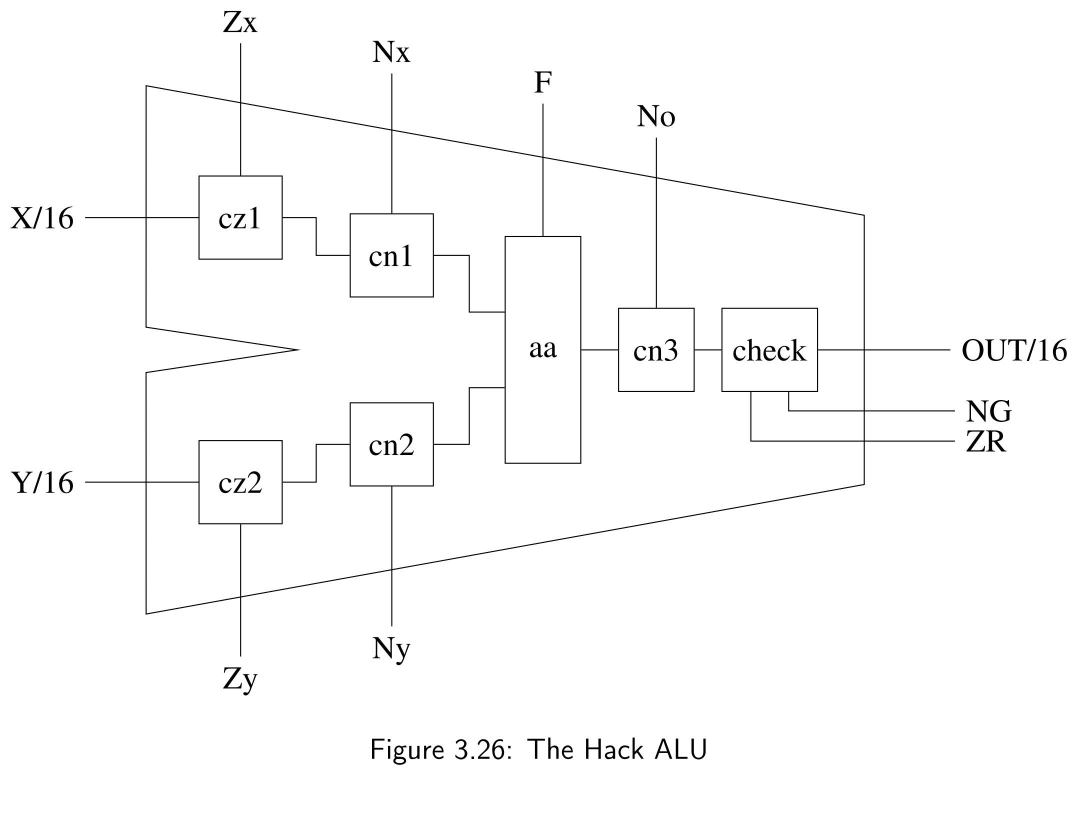


Figure 3.26: The Hack ALU

To construct the ALU in SystemC, two boolean input buses *X/16* and *Y/16* as well as six input signals (*ZX*, *ZY*, *NY*, *NY*, *F*, and *NO*) were used to produce the output bus *OUT/16* and two output signals *ZR* and *NG*. The digital logic circuits utilized included two *Controled_zero16*s (named *cz1* and *cz2*), three *Controlled_not16* chips (named *cn1*, *cn2* and *cn3*), an *AndORadd16* unit (referred to as *aa*) and a *check16* circuit (simply named *check*). Figure 3.26 provides a visual representation of the ALU's logical construction.

3.15 Constructing more hardware: a single-bit and 16-bit register

Next, we designed a single bit register and then built a larger, 16-bit register. To construct the single-bit register in SystemC, a single boolean input *IN* was fed in, and boolean input signals *LOAD* and *CLOCK* were also fed in. The *MUX* logic gate, as well as a *digital flip–flop* (commonly referred to as a "dff") were also used. Two internal signals *S* and *T* were used, and the output signal was *OUT*. See Figure 3.28 for a visual representation of the single-bit register compilation.

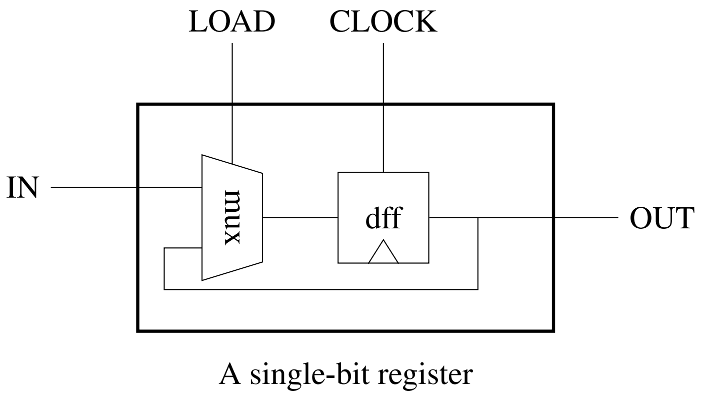


Figure 3.27: The internal composition of a single-bit register.

With the single–bit register constructed, we were able to combine sixteen together. The 16–bit register (also simply referred to as *register*) consists of 16 single–bit registers (*b0-b15*), a 16-bit input bus, a *LOAD* bit, and a *CLOCK* bit. Its full logical layout is shown in Figure 3.28.

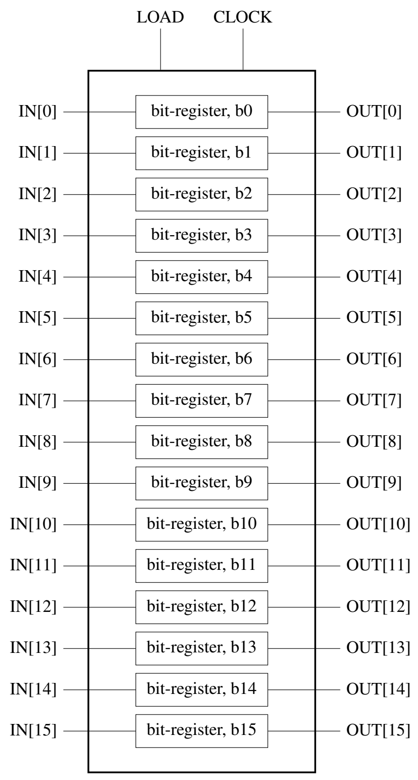


Figure 3.28: A 16-bit register is capable of holding the 16-bit input values HACK uses to operate.

3.16 Let's store some memory

Having constructed the 16-bit *register*, we now can construct the random access memory (RAM). The *RAM8* unit allows us to address eight words of memory. Once we have chosen what address we want, we can either read information from the location or write information to the location. We use recursive ascent to build larger memories. The following demonstrate the recursive ascent approach.

To construct the RAM8, eight 16-bit *registers*, a *D-mux_8*, and a *Mux8way16* were utilized. Additionally, three boolean signals (*LOAD*, *CLOCK*, and *ADDR*) were needed. The circuit is shown in Figure 3.29.

To build the *RAM64*, eight *RAM8*'s, a *D-mux_8*, and a *Mux8way16* are utilized. A boolean input array *IN/16* is routed into the *RAM64* circuit together with a *LOAD* and *CLOCK* bit. The circuit is shown in Figure 3.30.

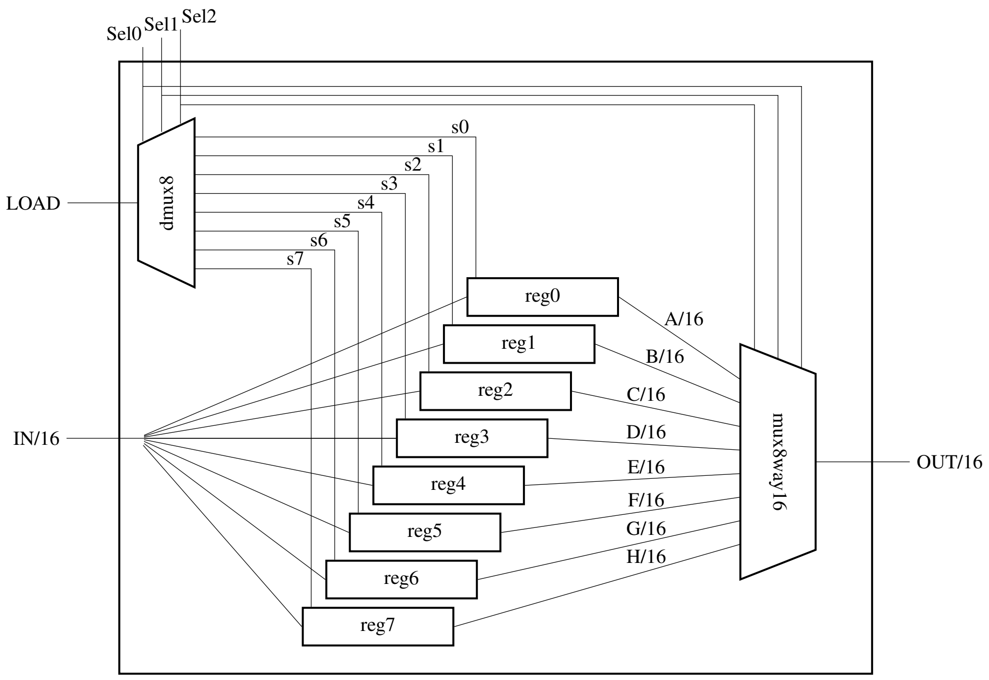


Figure 3.29: RAM8

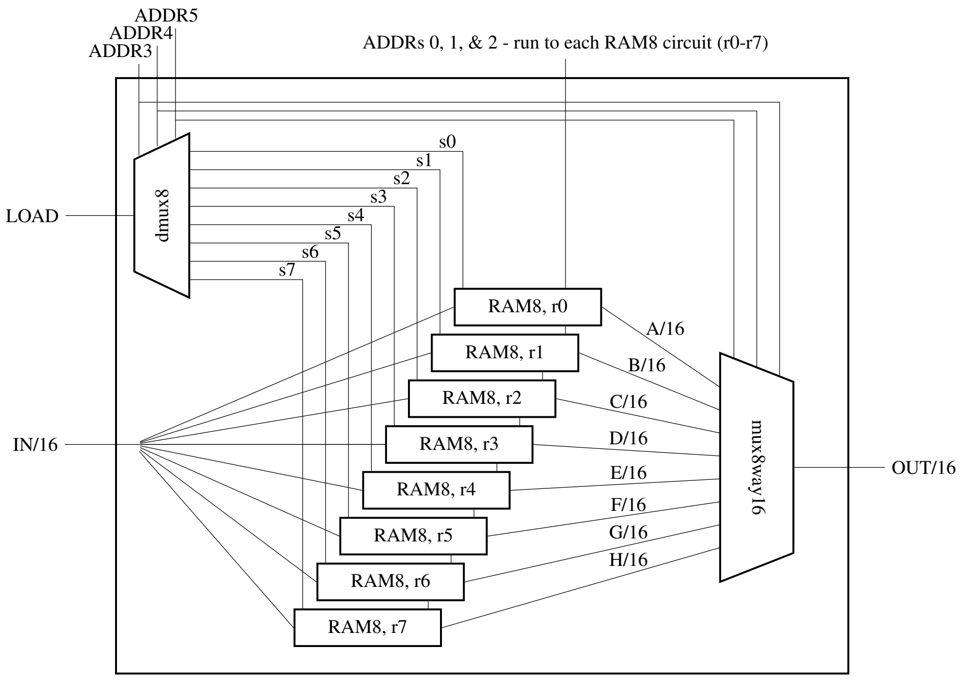


Figure 3.30: RAM64

3.17 Making more and more memory

With the *RAM64* now implemented, the next step was to continue to increase memory size. We did so, building the RAM512 and ultimately the RAM4K. While building the RAM4K we found the memory resources of our own computer to be extremely strained while running the simulation, ultimately being forced to use a machine with 16–GBytes of RAM (the simulation of the RAM4K used 9–GBytes). Figures 3.31 and 3.32 illustrate the RAM512 and RAM4K. Because the memory requirement became so intense, we elected to stop constructing larger memory sizes.

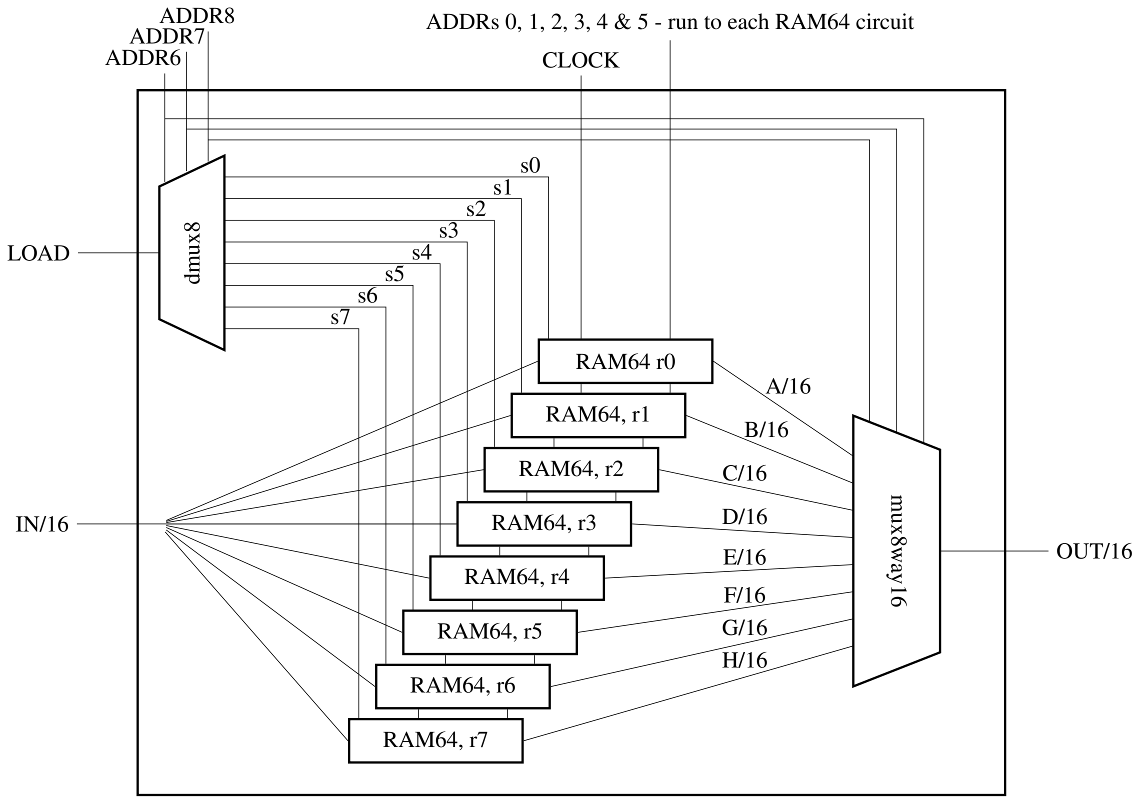


Figure 3.31: RAM_512

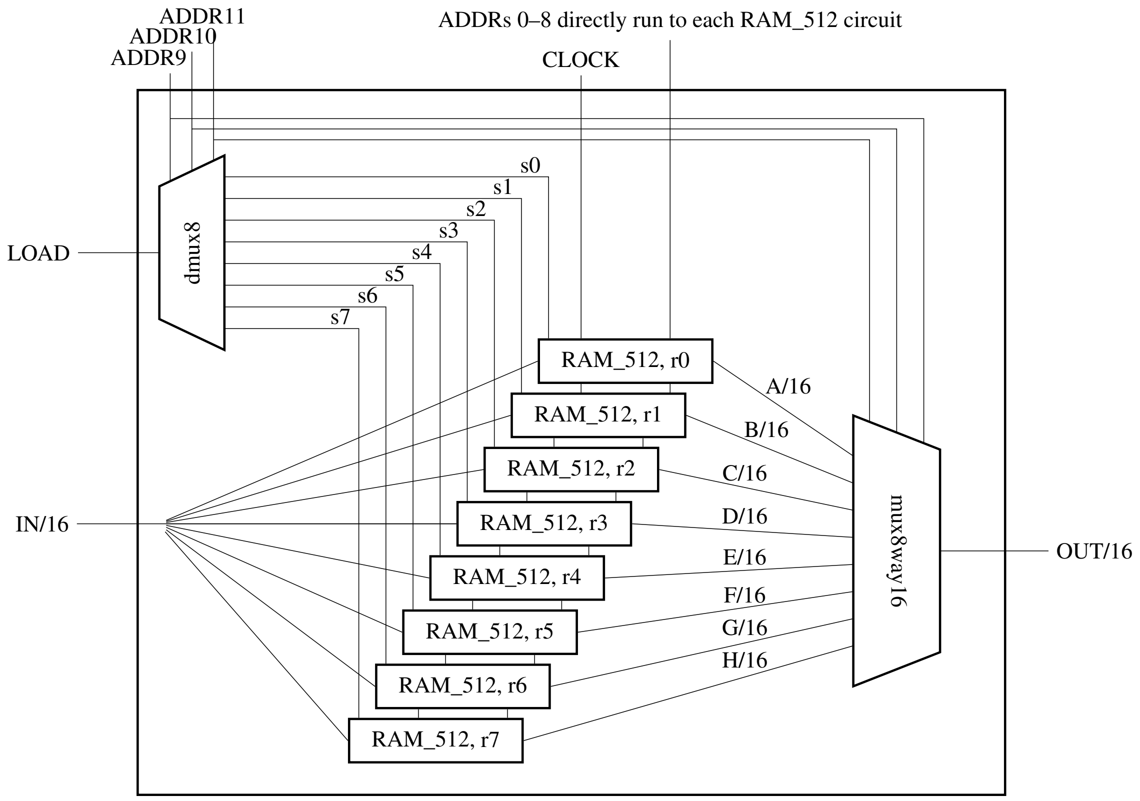


Figure 3.32: RAM4K

3.18 The Program Counter

Now that we have constructed memory, our next task as we work towards building the HACK machine is to create a program counter. The *program counter* accepts as input a boolean bus *IN/16* and boolean signals *LOAD*, *INC*, *RESET*, and *CLOCK* and produces a 16-bit boolean output *OUT/16*. The program counter utilizes a *mux16*, *controlled_zero16*, one 16-bit *register*, and a *INC_16*. There are four internal boolean buses and two internal boolean signals inside the *program counter*. The logic design of the *program counter* is shown in Figure 3.33.

3.19 Now it's time to jump

An integral part of any CPU is having the ability to jump while running a program. Thus, we next built the *JumpDetermination* circuit. There are three bits in each 16-bit instruction

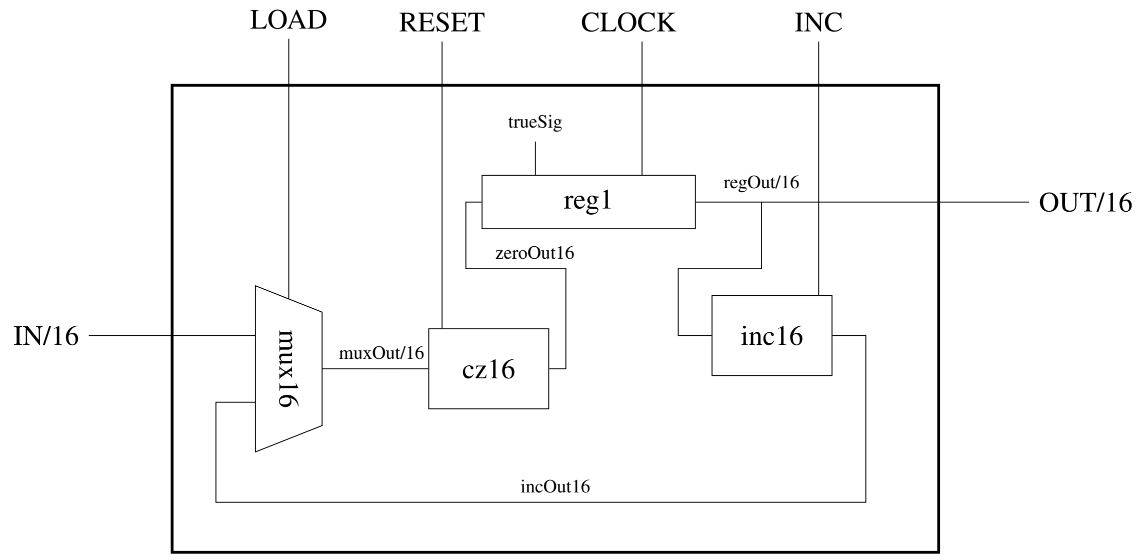


Figure 3.33: The "Program Counter"

that specify whether or not to jump. These three bits are named *J1bit*, *J2bit*, and the *J3bit*. You must also know if the instruction is a C-instruction; thus, we have the *isCinstruction* bit. The *NG* bit (informing us if the number is negative) and *ZR* bit are also needed.

The *JumpDetermination* requires a total of 25 *AND* gates, seven *NOT* gates, and seven *OR* gates. The logic design is illustrated in Figure 3.34.

3.20 Finally, the CPU

We are finally ready to build the *central processing unit*, or *CPU*. The HACK computer, a von Newmann machine, stores data and instructions in memory. The *machine language* is called the HACK machine language. A comprehensive explanation of the HACK machine language is provided in Chapter 4 of *The Elements of Computing Systems* [12]. If the MSB is set to zero (0), the bits are interpreted as an address or data. If the MSB is set to one, the bits are interpreted as an instruction [12].

Instructions allow the CPU to know what operations the user wants performed - logical, arithmetic, etc. There are four main fields of HACK instructions (known as *C-instructions*). Figure 3.35 shows the fields. Seven bits (*a*, *c1*, *c2*, *c3*, *c4*, *c5*, and *c6*) instruct the ALU what operation to preform. The destination bits (*d1*, *d2*, and *d3*) inform the CPU where to

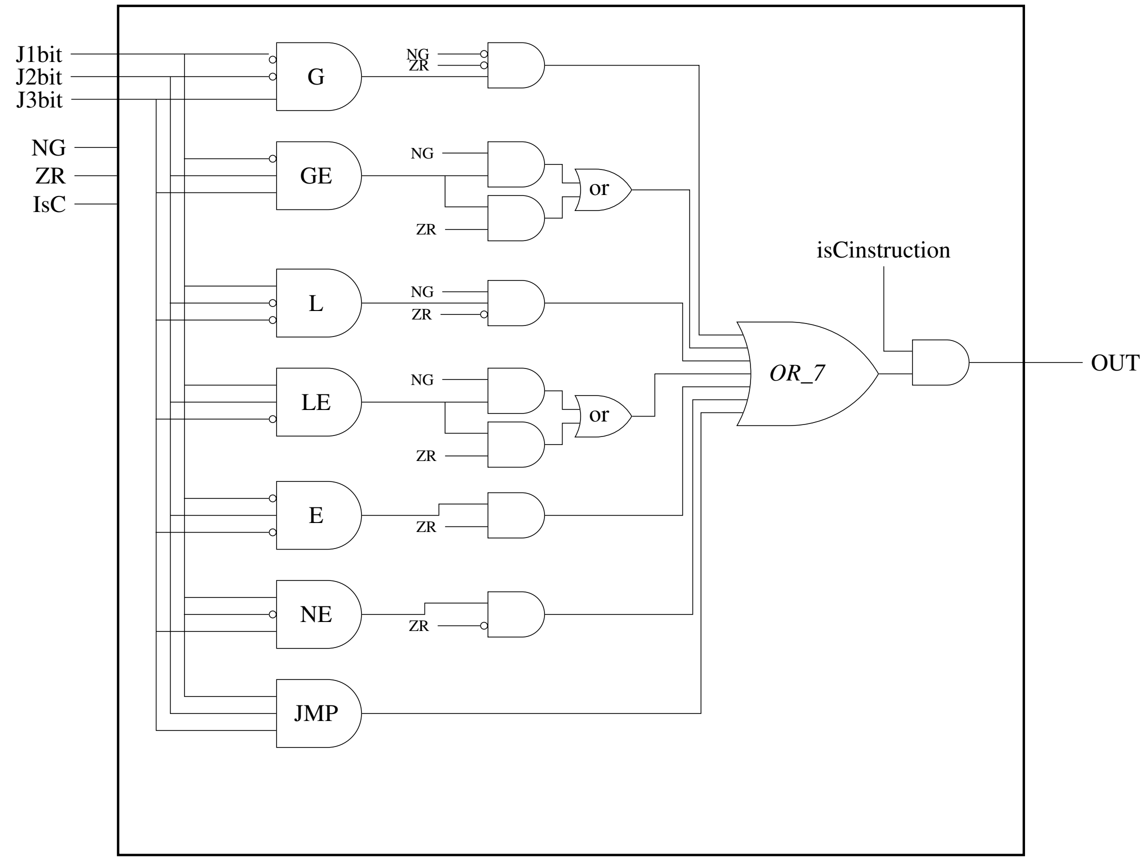


Figure 3.34: JumpDetermination

send the ALU's output, and the jump bits (*j1*, *j2*, and *j3*) determine whether a "jump" is needed.

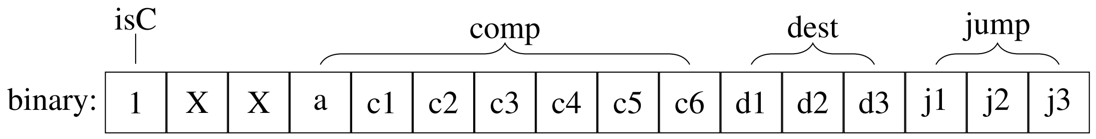


Figure 3.35: The C-Instruction's four fields.

The CPU evaluates the MSB, checking to see wether the bits represent a *C-instruction* or *A-instruction*. The CPU *fetches* (i.e., *reads*) a word from the instruction memory, decodes it, and executes the specified instruction [12].

Two figures illustrate the function of the CPU and its internal components. Figure 3.36 shows the inputs and outputs of the CPU at a high level of abstraction. Figure 3.37 shows the internal components of the CPU and the logical connections needed.

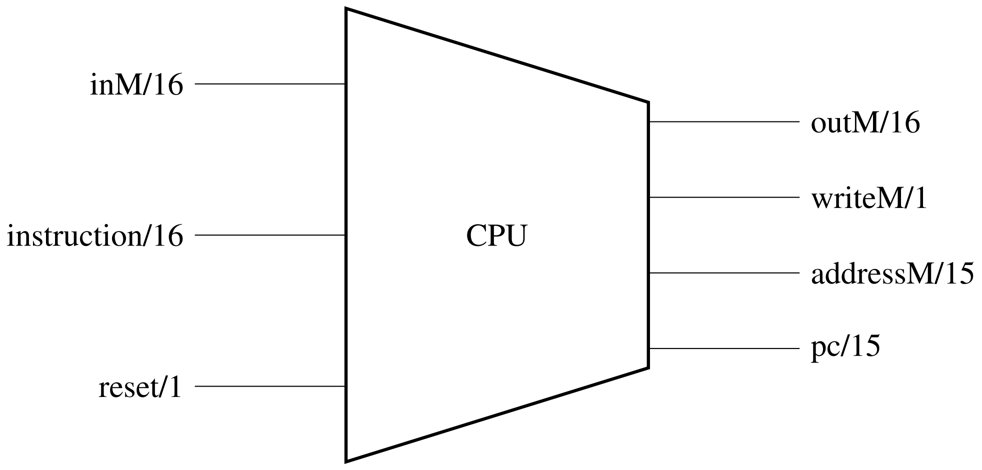


Figure 3.36: A high–level diagram of the CPU showing both inputs and outputs.

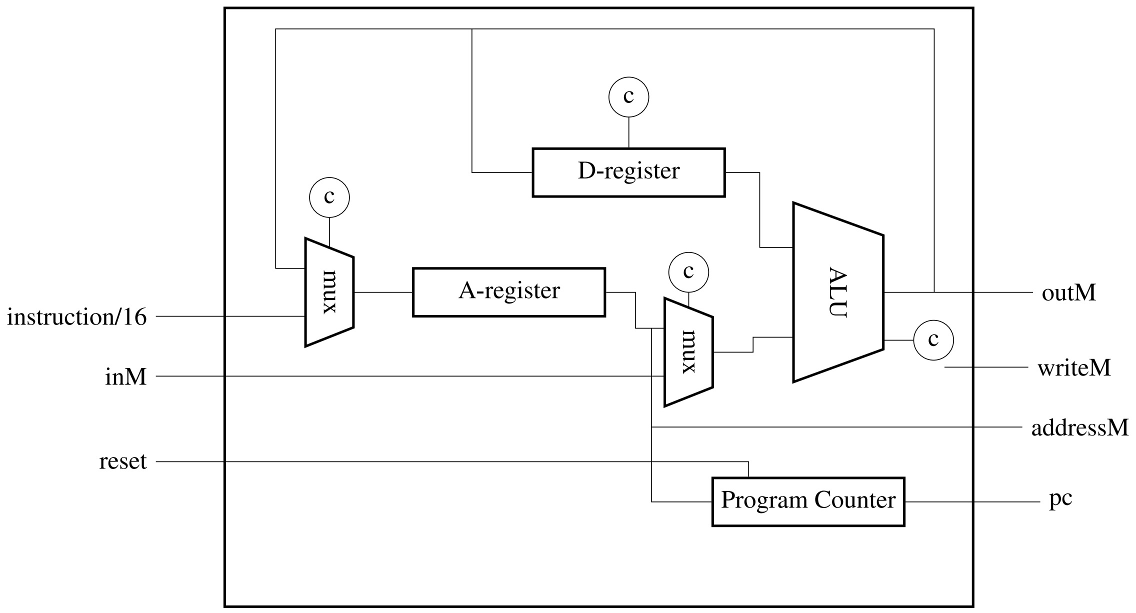


Figure 3.37: A low–level diagram of the CPU shows the required internal logical circuits. Each circled c refers to control logic.

3.21 And now the computer

To accommodate SystemC constraints, we diverted from Nisan and Schocken in the following respects. First, the size of the memory (both instruction memory and data memory) was limited to 64 addresses (aka RAM64). When simulating HACK in SystemC with memory sizes larger than RAM64, simulation required much more than 4-Gbytes. However, the RAM64 size was sufficient to demonstrate small programs. The second deviation from the Nand2Tetris model was the lack of a pre-built ROM (Read-Only Memory). To address this problem, we used a RAM64 logic circuit as instruction memory.

The HACK computer has three main components: an instruction-memory, the CPU, and data-memory. The first program we tested on the HACK computer was an addition program. This simple program adds the numbers two and three together. If all logic gates have been assembled and wired together correctly, the output will be five. This program requires six instructions; they are the following:

```
//@2
0000000000000010
//D=A
1110110000010000
//@3
0000000000000011
//D=D+A
1110000010010000
//@0
0000000000000000
//M=D
1110001100001000
```

To make this program work, we assemble the computer as depicted in Figure 3.38, then to load the program into instruction memory, we load each instruction one-by-one, making sure that the SET bit is enabled and the GET bit disabled during the loading process. Next we run the program for a predetermined, fixed number of clock cycles, with SET and GET disabled. Finally, we read the contents of the data memory at address zero. This requires one additional clock cycle with the GET bit enabled and the SET bit disabled. At the completion of the simulation, the contents inside address zero equal five, exactly what we expected.

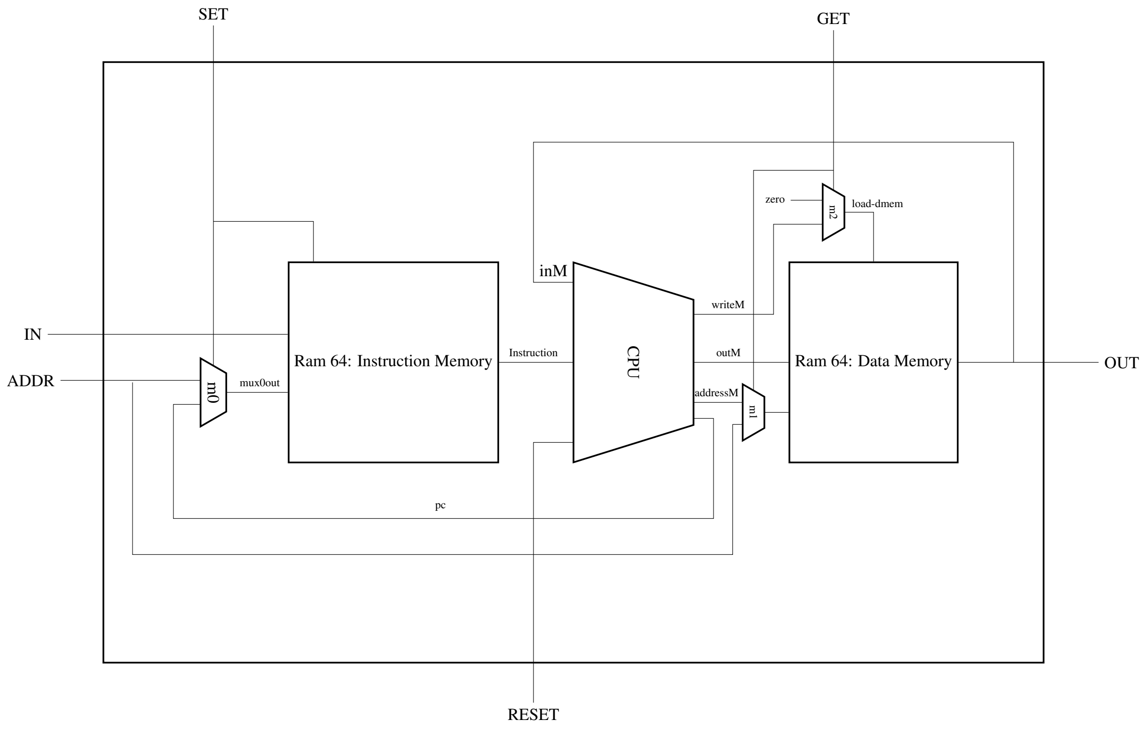


Figure 3.38: The HACK computer consists of three main parts: instruction memory, the CPU, and data memory.

CHAPTER 4: Experimental Setup and Results

Our thesis project both constructs a simple modern computer utilizing SystemC and facilitates the design and integration of policy enforcement circuitry. Having constructed the HACK computer described in the previous chapter and verified that it can correctly run a simple program, our next step was to run a more complex program and verify that the computer yielded a correct output. We chose to run the *multiply* program provided by Nisan and Schocken [12]. The instructions of the multiply program are provided below:

```
// This is the multiply program - it is a simple test program for multiplying
// two numbers.  We will initialize the instruction memory with this program.
// M[0] should contain the value 30 at the end of the program.

// @5
0000000000000101
// D=A
1110110000010000
// @R1
0000000000000001
// M=D
1110001100001000
// @6
0000000000000110
// D=A
1110110000010000
// @R2
0000000000000010
// M=D
1110001100001000
// @R0
0000000000000000
// M=0
```

```
1110101010001000
// @i
0000000000010000
// M=0
1110101010001000
// (LOOP)
// @R1
0000000000000001
// D=M
1111110000010000
// @i
0000000000010000
// D=D-M
1111010011010000
// @END
0000000000011010
// D;JLE
1110001100000110
// @R2
0000000000000010
// D=M
1111110000010000
// @R0
0000000000000000
// M=M+D
1111000010001000
// @i
0000000000010000
// M=M+1
1111110111001000
// @LOOP
0000000000001100
// 0;JMP
1110101010000111
```

```
// (END)
// @END
0000000000011010
// 0;JMP
1110101010000111
```

The multiply program consists of 28 instructions; thus, the *program length* is set at 28. Unlike the simple addition program in Chapter 3, the multiply program also contained "looping" via conditional jumps. Because of this feature, we had to increase the *max* value to allow enough cycle-iterations to complete the program. We chose to set max to a value of 128. At the conclusion of the 128 clock cycles, the value of M[0] was thirty, exactly what was expected.

With the outputs of both programs (*addition* and *multiply*) yielding correct results, our confidence in the creation of the HACK machine was sufficient to move to the second portion of our project—the design of policy enforcement circuitry. To demonstrate built-in policy enforcement capabilities, we designed an *integrity checker*.

4.1 Creating the Integrity Checker

The *integrity checker* protects the HACK machine's *data memory* against unauthorized modifications. Our simple demonstration assumes that the first four lines of instruction code come from an "untrusted" source; thus, we want to prevent the instructions from writing to unauthorized memory locations. We chose to allow untrusted code to write to the first sixteen *data memory* addresses, but to deny the untrusted code from writing to any other addresses.

We constructed the *integrity checker* using four OR-gates, five AND-gates, and six NOT-gates. The program counter's second, third, fourth, and fifth bits are fed in to check if the instruction resides in the secure or insecure portion of the code. If the *integrity checker* (i.e., "checker") detects the value in the program counter to be less than four, it will deny writes to data addresses greater than or equal to sixteen. If any of the values of pc2, pc3, pc4, or pc5 is one, this indicates that the instruction comes from the trusted portion of the code; thus, the program is allowed to write to any location in data memory. Figure 4.1 illustrates the logical layout of the integrity checker described above.

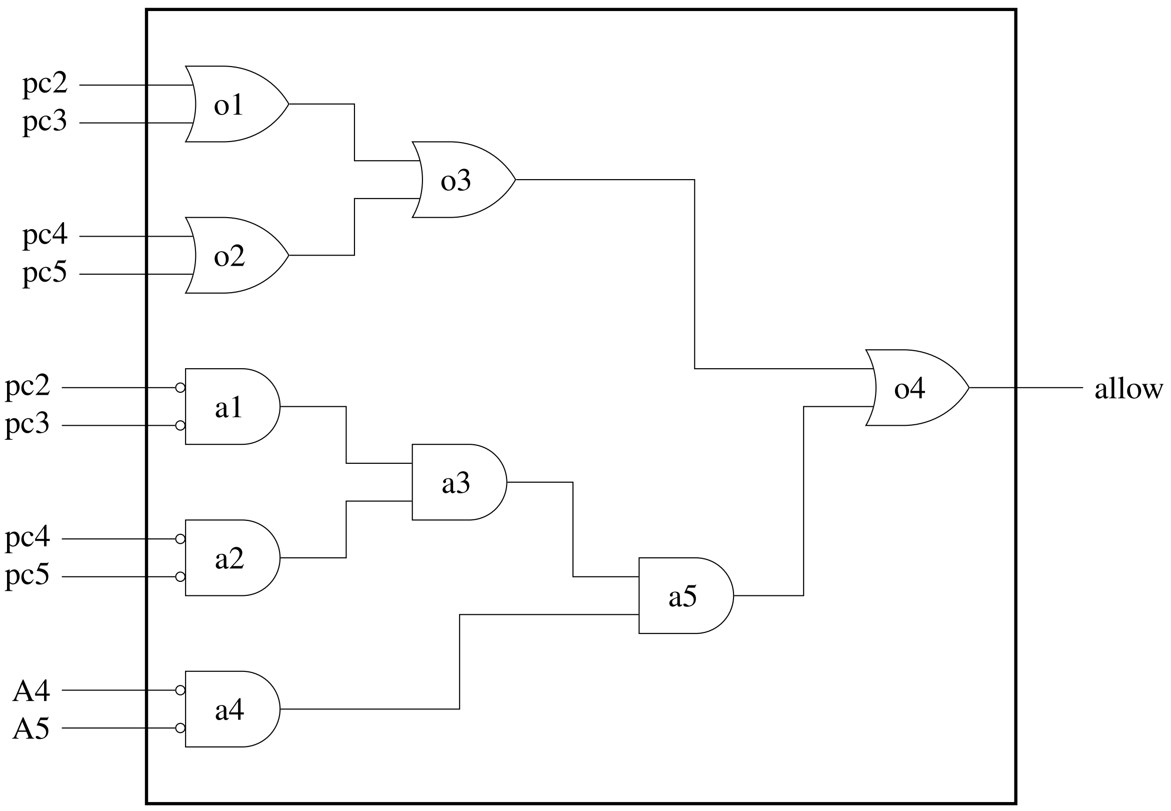


Figure 4.1: The Integrity Checker.

4.2 Incorporating the Integrity Checker in the HACK computer design

Integrating the *integrity checker* into the HACK computer design is straightforward, merely affecting the "writeM" bit, which proceeds from the CPU and is fed into *mux2*. The mux2 gate accommodates the GET bit. The output of the mux2 gate is then "anded" with the output of the checker. If the writeM bit is enabled and the *allow*-bit proceeding from the checker is also enabled, the output of the CPU, *outM*, will be written into *data-memory* at the specified address. Conversely, if the *integrity checker* detects that an instruction in the untrusted portion of the code is trying to write to a prohibited address, the allow-bit will be *disabled*, and writing to data-memory will be denied.

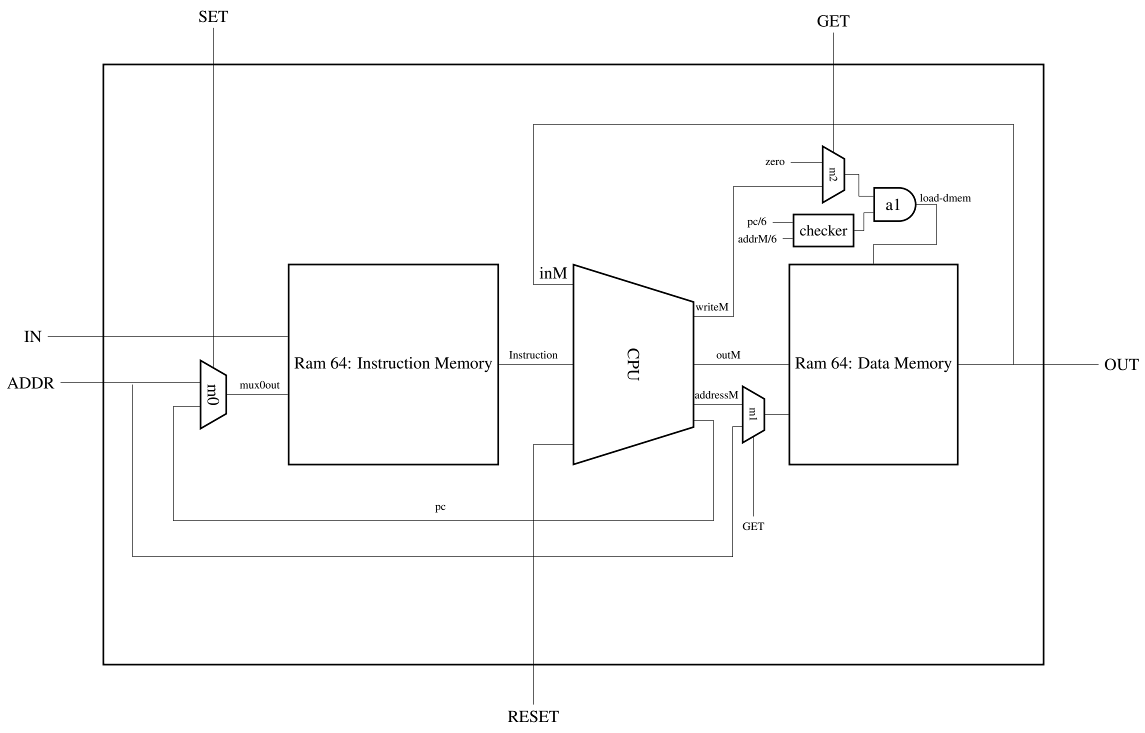


Figure 4.2: The HACK computer with the "INTEGRITY CHECKER" installed.

4.3 The Test

To demonstrate that the *integrity checker* correctly prevents writes to unauthorized locations, we created and ran the following test program:

```
//(UNTRUSTED)
//@R0
0000000000000000
//M=1
1110111111001000
//@16
0000000000010000
//M=1
1110111111001000

//(TRUSTED)
//@7
```

```
0000000000000111
//D=A
1110110000010000
//@R0
0000000000000000
//M=D
1110001100001000
//@16
0000000000010000
//M=D
1110001100001000
//(END)
//@END
0000000000001010
//0; JMP
1110101010000111
```

As previously discussed, the first four instructions are considered untrusted. The next eight instructions are considered trusted. To validate the functionality of the checker, we first ran the test code in the HACK machine without installing the integrity checker. After two clock-cycles, the program should write the value of one (1) into M[0] (data-memory address zero). After four clock-cycles, the program will write a value of one (1) into M[16]. After eight clock-cycles, the program will write a value of seven (7) into M[0], and after ten clock-cycles the value of seven will also be written into M[16]. Running the program yielded the expected results, as shown in Table 4.1. For this program, we set *max* to twenty-four.

Memory Address	Clock-cycles	Data-Values
M[0]	2	1
M[16]	4	1
M[0]	8	7
M[16]	10	7

Table 4.1: Memory values after running the "test" program in the HACK machine without installing the integrity checker.

We next ran the "test" program on the HACK computer with the *integrity checker* installed. Untrusted code can still write to M[0-15] of data-memory, but not to data-memory locations greater than or equal to sixteen. As shown in Table 4.2, M[0] has a value of one after two clock-cycles, M[16] remains uninitialized after four clock-cycles, and the values of M[0] and M[16] will be set to seven by the trusted instructions as before.

Memory Address	Clock-cycles	Data-Values
M[0]	2	1
M[16]	4	uninitialized
M[0]	8	7
M[16]	10	7

Table 4.2: Memory values after running the "test" program with the checker installed.

4.4 Impact on system performance

To measure the impact of the *integrity checker* on system performance, we run the test program both with and without the integrity checker installed. We measure simulation time using the UNIX 'time' command. The timing results are listed in Tables 4.3 and 4.4, with *max* set to a value of twenty-four. Installing the integrity checker has minimal impact on system performance (Averages with checker installed – Real: 2.608, User: 2.276, and System: 0.285, versus averages without checker installed – Real: 2.601, User: 2.275, and System: 0.284).

Tests run without security mechanism	Test One	Test Two	Test Three	Test Four
Real	2.616	2.603	2.602	2.581
User	2.279	2.278	2.263	2.280
System	0.286	0.277	0.289	0.283

Table 4.3: Timing results of running test program without inclusion of the *integrity checker*.

Tests run with security mechanism	Test One	Test Two	Test Three	Test Four
Real	2.608	2.609	2.620	2.598
User	2.273	2.275	2.290	2.269
System	0.289	0.285	0.286	0.282

Table 4.4: Timing results of running test program with the inclusion of the *integrity checker*.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5: Conclusion and Future Work

The growth in system complexity enabled by Moore's Law poses major challenges for designers to ensure correct operation, security, fault tolerance, and other key properties. Since electronic design automation (EDA) tools (e.g., logic synthesis) first appeared decades ago, the complexity of both hardware and software has risen dramatically. To tackle this problem, high-level synthesis (HLS) is a design methodology being embraced by major chip manufacturers, including Xilinx, which incorporates HLS into its Vivado design software suite. Though Vivado does not use SystemC, it uses another C–like language to express both hardware and software. HLS is related to electronic system-level (ESL) design, which allows a chip designer to express an algorithm in a high-level language, and the design tools automatically translate the high-level specification into a cycle-accurate system. Since designers wishing to build a secure system must be able to fully comprehend the system and have mastery over the design tools that practitioners are using, HLS has the potential to facilitate trustworthy system development by helping designers efficiently express hardware and software components for computation, communication, security, reliability, etc. Also, since modeling languages like SystemC allow designers to express both hardware and software, they facilitate co-simulation of hardware and software in the same simulation environment, which offers the potential of greater efficiency than traditional methods.

To explore the application of HLS to trustworthy system development, this thesis applied a HLS design approach to a simple 16-bit computer, implementing it in the popular SystemC modeling language. With only a C++ compiler and the SystemC library, we were able to design and simulate the CPU, instruction memory, and data memory. Compiling the C++ code and linking with the SystemC library generates an executable, and running the executable performs a simulation of the hardware. While simulating the computer's hardware, we were able to run programs in the machine language of the CPU on the simulated hardware. We had to reduce the size of our simulated computer's memory due to the well-known problem of fine-grained simulations requiring large amounts of memory as system complexity increases. We integrated a simple memory integrity policy enforcement mechanism, also designed in SystemC, into our design. Our simulation results show that the mechanism correctly enforces the integrity policy and imposes minimal impact on overall system performance.

We see many opportunities for future work. For example, it would be worthwhile to explore more complex processor designs and more sophisticated security policy enforcement mechanisms. It would also be useful to learn more about the capabilities of production-grade, tape-out tools like Vivado HLS (e.g., modeling a system with Vivado and prototyping it on an inexpensive FPGA board, such as the Basys 2 board from Digilent). We would also like to overcome the technical hurdles to more fully implement the 16-bit design, including a display, keyboard, and larger memory. We would like to perform more optimizations on our 16-bit design to make it more efficient and to compare its implementation in SystemC against the same design expressed in traditional HDL. We would like to become more proficient in SystemC so that we can more efficiently and elegantly express circuits. We would also like to express our design in other languages, environments, and tool flows. Finally, the broader impact of our work aims to make it easier for Computer Scientists to leverage custom hardware's benefits.

References

[1] M. Tehranipoor, H. Salmani, X. Zhang, W. Xiaoxiao, R. Karri, J. Rajendran, and K. Rosenfeld, "Trustworthy hardware: Trojan detection and design-for-trust challenges," *Computer*, vol. 44, no. 7, pp. 66–74, 2011.

[2] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware Trojans," *Computer*, vol. 43, no. 10, pp. 39–46, 2010.

[3] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[4] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: Orthogonalization of concerns and platform-based design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, no. 12, pp. 1523–1543, Dec 2000.

[5] O. Shacham, O. Azizi, M. Wachs, W. Qadeer, Z. Asgar, K. Kelley, J. Stevenson, S. Richardson, M. Horowitz, B. Lee, A. Solomatnikov, and A. Firoozshahian, "Rethinking digital design: Why design must change," *Micro, IEEE*, vol. 30, no. 6, pp. 9–24, 2010.

[6] P. Coussy and A. Morawiec, *High-Level Synthesis*. Heidelberg, Germany: Springer, 2010.

[7] L. A. D. Bathen and N. Dutt, "PoliMakE: A policy making engine for secure embedded software execution on chip-multiprocessors," in *Proceedings of the 5th Workshop on Embedded Systems Security*. ACM, 2010, p. 2.

[8] L. A. D. Bathen and N. D. Dutt, *PHiLOSoftware: A Low Power, High Performance, Reliable, and Secure Virtualization Layer for On-Chip Software-Controlled memories*. PhD Dissertation, Department of Computer Science, University of California, Irvine, Irvine, CA, 2012.

[9] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sept 1975.

[10] CNSS Secretariat, "Supply Chain Risk Management (SCRM)," CNSS Directive 505, Fort Meade, MD, March 2012.

[11] J. Follett. (2008). Cisco channel at center of FBI raid on counterfeit gear. [Online]. Available: http://www.crn.com/news/networking/207602683/cisco-channel-at-center-of-fbi-raid-on-counterfeit-gear.htm

[12] N. Nisan and S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles*. Cambridge, MA: MIT Press, 2005. [Online]. Available: http://www.nand2tetris.org

[13] C. C. Lin. (2003). What's a multiplexer (and a demultiplexer)? [Online]. Available: http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Overall/mux.html

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California